METHOD AND APPARATUS FOR LINKING CONVERTED APPLET FILES WITHOUT RELOCATION ANNOTATIONS

BACKGROUND

1. Field of the Present Invention

The present invention generally relates to the field of smart cards and more particularly to an improved smart card and method for determining the boundaries of method bodies within converted applet files loaded into a smart card.

2. History of Related Art

Most consumers are familiar with and use credit cards, debit cards, automatic teller machine (ATM) cards, stored value cards, and the like. For many types of transactions, however, the current trend is away from these types of cards and into a class of devices generally referred to as smart cards. A smart card is a plastic, credit-card sized card that includes an electronic device (chip) embedded in the card's bulk plastic. Rather than only employing a magnetic strip to store information, smart cards employ a microprocessor and a memory element embedded within the chip.

Because they have a chip, smart cards can be programmed to operate in a number of varied capacities such as stored value cards, credit cards, debit cards, ATM cards, calling cards, personal identity cards, critical record storage devices, etc. In these varied capacities, a smart card may be designed to use a number of different application programs. Smart cards are compliant with Standard 7816 Parts 1-10 of the International Organization for Standardization (ISO), which are incorporated by reference herein and referred to generally as "ISO 7816."

Initially, application program development for smart cards was essentially proprietary to the smart card manufacturers or the smart card issuers. Smart card application development has, however, evolved over recent years so that it is no longer proprietary. Through the adoption of open architectures for application development, it is now possible to develop applications that can run on smart cards from different manufacturers, on other devices for the storage of data (i.e., storage devices), or other resource constrained devices that, like smart cards, have small amounts of available memory. Java Card technology is an example of one such open development architecture. It uses the Java™ programming language and employs the

Java Card runtime environment (JCRE). The JCRE conforms to ISO 7816 and defines a platform on which applications written in the Java programming language can run on smart cards and other resource-constrained devices. Applications written for the JCRE are referred to as applets and must conform with the Java Card 2.1 Virtual Machine Specification (as revised from time to time) published by Sun Microsystems, Inc., which is incorporated by reference herein and referred to generally as the "JCVM Specification."

The process of loading an applet onto a smart card for execution requires that the source code of the applet first be converted into a corresponding binary representation of the classes making up the applet. This corresponding binary representation is referred to as a CAP file (converted applet file) and is the file format in which applications are loaded onto smart cards utilizing the JCRE. The CAP file is typically loaded as a block of bytes occupying contiguous space in the non-volatile read/write memory of the smart card by an installer module located on the smart card.

A CAP file consists of a set of components each of which defines differing elements or aspects of the contents of the CAP file. One such component is the Method Component which defines each of the methods (i.e., procedures or routines associated with one or more classes) declared in the package that makes up the CAP file. The following are among the items included in the Method Component: (1) a size item which indicates the number of bytes in the Method Component, (2) a handler count item which indicates the number of exception handler entries in the exception handler array, (3) an exception handlers item which provides relevant information for each exception handler (including a starting offset and a length for indicating the range of bytecodes for which the exception handler is active (i.e., a "catch range") and further including the starting address of the exception handler), and (4) a methods item which defines each of the variable length methods (i.e., method bodies) declared in the package making up the CAP file with each such method body containing a method header followed by the bytecodes (i.e., instructions) that implement such method body. While each method header contains relevant information defining the requirements for the operand stack, associated parameters being passed to the method body, and local variables for the method body, the method header does not specify the size of the associated method body.

The bytecodes within each method body typically contain operands consisting of various symbolic or unresolved code references which must be resolved prior to

execution. The process of resolving these particular operands is generally referred to as linking or resolution and involves looking-up the symbolic reference in a corresponding table present in memory (constant pool) or other storage device or calculating the unresolved relative code reference and replacing the reference with the actual memory address or an internally accessible symbolic reference at which the particular command, function, definition, etc. is stored. The terms "resolve," "resolution," "resolving," and "linking" are used throughout to broadly describe the foregoing process of replacing the unresolved code or symbolic reference within the code or data structure with an internally accessible symbolic reference or actual memory address.

One method utilized by the prior art to implement the resolution process is to access a list of bytecode offsets into each method body, which list designates the applicable bytecodes within each method body requiring resolution (i.e., relocation annotations). This list of bytecode offsets is provided in the form of another component in the CAP file referred to as the Reference Location Component. The Reference Location Component is required by the prior art because the boundaries of each method body (i.e., the beginning and the end of each method body) are not specified in the Method Component. The Reference Location Component is used solely for the resolution process and is not referenced by any other component in a CAP file.

A second method utilized by the prior art to implement the resolution process without accessing the list of relocation annotations contained in the Reference Location Component is to access information contained in another component of the CAP file referred to as the Descriptor Component. The Descriptor Component contains sufficient information to permit parsing and verification of all elements of the CAP file. Consequently, the Descriptor Component contains information on the boundaries of each method body so as to permit parsing and examination of the bytecodes contained within each method body. Because of its lengthy size, however, the Descriptor Component is an optional component of the CAP file, and as such, is not always available.

It would be beneficial to implement an apparatus and method for the efficient linking of CAP files that does not rely upon relocation annotations (such as those provided in the Reference Location Component), but rather which enables the contiguous bytecodes making up the methods item of the Method Component to be

examined (and if necessary, their operands resolved) without reliance upon the Descriptor Component.

BRIEF DESCRIPTION OF THE DRAWINGS

The structure and operation of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

FIG. 1 shows an exemplary smart card;

FIG. 2 conceptually illustrates the microprocessor with integral memory element **115** portion of module **105** of the exemplary smart card of FIG. 1 in some additional detail;

FIG. 3 conceptually illustrates a portion of non-volatile read/write memory **230** loaded with the method bodies for a Method Component;

FIG. 4 is a flowchart further illustrating one aspect of the present invention;

FIG. 5 is a flowchart further illustrating one embodiment for determining the farthest logical return and resolving applicable relocation references depicted in FIG. 4; and

FIG. 6 is a flowchart further illustrating one embodiment for examining the exception handler code depicted in FIG. 4.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description presented herein are not intended to limit the invention to the particular embodiment disclosed. On the contrary, the invention is limited only by the claim language.

DETAILED DESCRIPTION OF THE INVENTION

Generally speaking the present invention contemplates a smart card and method in which the smart card may be configured to receive computer code such as converted applet files with unresolved references within the method bodies for such files and examine each bytecode within the method bodies (and if applicable, resolve their operands) in a sequential manner and without relying upon relocation

annotations such as those provided in a Reference Location Component. Throughout the description and the drawings, elements which are the same will be accorded the same reference numerals.

FIG. 1 shows an exemplary smart card. Smart card **100** is roughly the size of a credit card and includes a module **105** with conductive contacts **110** connected to a microprocessor with an integral memory element **115**. Conductive contacts **110** interface with a terminal to typically provide operating power and to electrically transfer data between the terminal and the smart card. Other embodiments of a smart card may not include conductive contacts **110**. Such "contactless" smart cards may exchange information and may receive power via proximately coupling, such as magnetic coupling, or via remote coupling, such as radio communication. Smart card **100** is compliant with ISO 7816.

The microprocessor with an integral memory element **115** of FIG. 1 is shown in additional detail in FIG. 2. Microprocessor **115** includes central processing unit (CPU) **205**. Microprocessor **115** is associated with a memory element **215**. The "memory" may be formed on the same integrated circuit as microprocessor **115** or may be formed on a separate device.

Generally, the memory element **215** may be configured to include random access memory (RAM) **220**, read only memory (ROM) **225**, and non-volatile read/write memory **230**. Read only memory **225** may be configured to include installer module **210**. In an alternative configuration, RAM **220**, ROM **225**, and non-volatile read/write memory **230** are not located in the same memory element **215**, but rather, in some combination of separate electronic units.

FIG. 3 conceptually illustrates a portion of non-volatile read/write memory **230** loaded with variable length method bodies 1 **335**, 2 **340**, and Y **345** of the Method Component for a CAP file loaded into non-volatile read/write memory **230** by installer module **210.** Method Headers 1 **305**, 2 **310**, and Y **315** contain relevant information for method bodies 1 **335**, 2 **340**, and Y **345**, respectively. Further, the bytecodes for method bodies 1 **335**, 2 **340**, and Y **345**, are depicted by **320**, **325**, and **330**, respectively.

FIG. 4 depicts a flow diagram illustrating an embodiment of a method **400** of the present invention. The embodiment generally involves examining each of the

bytecodes making up the methods item of the Method Component, determining the starting point and ending point of the bytecodes containing instructions for each method body (including applicable exception handler code), and resolving each of the instruction bytecodes containing operands requiring resolution. In the depicted embodiment, using the applicable size information from the Method Component, a Start Pointer and an End Pointer are set to the start (i.e., method header of the first method body) and to the end of the Method Component, respectively **410**. Using relevant size information from the method header of the applicable method body, the Start Pointer is incremented to the first bytecode of the applicable method body **415**. A determination is made as to whether the Start Pointer is less than the End Pointer **420**. If the Start Pointer is less than the End Pointer (i.e., the end of the Method Component has not been reached), then the farthest logical return ("FLR") for the applicable method body (i.e., end of the method body) is determined **425** and each bytecode up to and including the FLR is examined and its corresponding operands resolved as necessary **430**.

The exception handler array is then examined to determine if one or more exception handler items are present for the bytecode range of the applicable method body **435**. If an exception handler item for the bytecode range of the applicable method body is present, then the FLR for the exception handler item (i.e., end of the exception handler) is determined **440** and if the FLR for the exception handler code is greater than the previously determined FLR (i.e., the exception handling code is contained at the end of the method body), each bytecode up to and including the FLR for the exception handling code is examined and its corresponding operands resolved as necessary **445**. If an exception handler item for the bytecode range of the applicable method body is not present or (if applicable) following the determination of the FLR for the exception handler item and resolution of applicable bytecode operands for the exception handler item, the Start Pointer is incremented to the first bytecode following the FLR **450** (which will be a method header if a subsequent method body follows), the Start Pointer is skipped over the method header **415**, and the Start Pointer is once again examined to determine if it is less than the End Pointer **420**. If the Start Pointer is equal to or greater than the End Pointer **420**, then the bytecodes in each of the method bodies of the Method Component have been traversed and resolved (where applicable) and the process is completed.

FIG. 5 depicts a flow diagram illustrating an embodiment of a method **500** for determining the FLR for the current method body and for resolving the operands (where required) for each bytecode of the current method body as depicted in items **425**, **430**, **440**, and **445** of FIG. 4.

Upon entering the method, a FLR Pointer is set to equal the Start Pointer **505**. As noted in the detailed description for FIG. 4 above, the Start Pointer is pointing to the first bytecode following the method header of the applicable method body. A determination is then made as to whether the Start Pointer is less than or equal to the FLR Pointer **510**. If the Start Pointer is less than or equal to the FLR Pointer, then a BC Pointer is set to point to the bytecode currently pointed to by the Start Pointer **520**.

A determination is made as to whether the bytecode pointed to by the BC Pointer is of a type that makes a forward call (i.e., forward jump instruction) **525**. It will be apparent to those skilled in the art that the instructions in the JCVM Specification having a hexadecimal opcode value of 60 – 70, 73, 75, and 98 – A8 are illustrative of bytecode types making a forward call. If the bytecode pointed to by the BC Pointer is of a type that makes a forward call, a temporary FLR Pointer is set to the location of the forward call **530**. A determination is then made as to whether the temporary FLR Pointer is greater than the current FLR Pointer **535**. If so, a logical return for the method body that is farther than the current FLR has been located and the FLR Pointer is set to equal the temporary FLR Pointer **540**.

Following the examination as to whether the temporary FLR Pointer is greater than the current FLR Pointer (and if applicable setting the FLR Pointer to equal the temporary FLR Pointer), the bytecode pointed to by the BC Pointer is examined to determine if the applicable operand(s) require resolution **545**. If the applicable operand(s) require resolution, then the bytecode pointed to by the BC Pointer is resolved **550**. Following the examination (and applicable resolution) of the bytecode pointed to by the BC Pointer, the Start Pointer is incremented to the next bytecode **555**.

A determination is then made as to whether the bytecode pointed to by the BC Pointer is a logical return (i.e., valid ending instruction) **560**. It will be apparent to those skilled in the art that the instructions in the JCVM Specification having a hexadecimal opcode value of 70, 77 – 7A, 93, A8 are illustrative of bytecode types

that are a logical return. If the bytecode pointed to by the BC Pointer is not a logical return, then a determination is made as to whether the Start Pointer is greater than the FLR Pointer **565**. If the Start Pointer is greater than the FLR Pointer **565**, then the FLR Pointer is set to equal the Start Pointer **570**.

If the bytecode pointed to by the BC Pointer is a logical return or (if applicable) following examination of the Start Pointer and (if applicable setting of the FLR Pointer to equal the Start Pointer), the Start Pointer is once again examined to determine if it is less than or equal to the FLR Pointer **510**. If the Start Pointer is greater than the FLR Pointer **510**, the FLR for the current method body (other than applicable exception handling code) has been determined, each bytecode up to and including the FLR has been examined (and if necessary, its operands resolved), and the process completed.

FIG. 6 depicts a flow diagram illustrating an embodiment of a method **600** for determining if exception handling code is present for the current method body as depicted in item **435** of FIG. 4 and if so, for determining the FLR for the exception handling code and for resolving the operands (where applicable) for each bytecode of the exception handling code as depicted in items **440** and **445** of FIG 4. Upon entering the process, a determination is made as to whether the exception handler array for the Method Component is empty **605**. If the exception handler array is not empty, the following process is conducted for each entry in the exception handler array: the catch range for the entry is determined **615** and a determination is made as to whether the bytecodes for the current method body just examined are within the catch range **620**. If the bytecodes for the current method body just examined are within the catch range of the applicable exception entry, an Exception Pointer is set to the start of the applicable exception handler code for the applicable exception entry **625** and a determination is made as to whether the Exception Pointer is greater than the FLR Pointer **630**. If the Exception Pointer is greater than the FLR Pointer, then exception handling code exists for the current method body beyond the existing FLR and the FLR for the exception handling code is determined and each bytecode of the exception handling code up to and including the FLR is examined (and if necessary, its operands resolved) **440** and **445**, respectively.

The process depicted in Fig. 6 is complete if the exception handler array is empty **605** or (if applicable) following the examination of each entry in the exception handler array to determine whether the bytecodes for the current method body just examined are within the catch range of the applicable entry **620**.

It should be appreciated that portions of the present invention may be implemented as a set of computer executable instructions (software) stored on or contained in a computer-readable medium. The computer readable medium may include a non-volatile medium such as a floppy diskette, hard disk, flash memory card, ROM, CD ROM, DVD, magnetic tape, or another suitable medium.

As introduced above, the term "smart card" was described with reference to the device shown in FIG. 1. While this example serves well for the explanations which followed, it should be noted that the present invention is broadly applicable to a class of resource-constrained devices having physical form factors which are different from the one illustrated in the example. For example, the present invention is readily adapted to Secure Interface Modules (SIMs) and Secure Access Modules (SAMs). SIMs and SAMs are physically smaller versions of the typical smart card and are typically used within telephones or other small spaces. The size, shape, nature, and composition of the material encapsulating or mounting the microprocessor and memory element are not relevant or limiting to the present invention. Thus, as used throughout, the term "smart card" is to be broadly read as encompassing any self-contained combination of microprocessor and memory element capable of performing a transaction with another device referred to as a terminal.

A person skilled in the art will appreciate that there are many alternative implementations of the invention described and claimed herein. For example, the embodiments described use various pointers, e.g., the Start, End, FLR, BC, and Exception Pointers. These pointers may be implemented using pointer data types provided by many programming languages and operating systems. Alternatively, they may be implemented as offsets from given locations. For example, in the latter alternative, a base location may be set at the start of a method body. Each of the pointers may then be implemented as offsets from that base location. Thus, the word "pointer" herein connotes a data element that provides a way, function, and result of specifying a location in a program or data structure so that access can be made to the

content at that location thereby allowing examination, comparisons, or other operations involving that content.

It will be apparent to those skilled in the art having the benefit of this disclosure that the present invention contemplates a smart card and method for the efficient linking of computer code that does not rely upon relocation annotations. It is understood that the forms of the invention shown and described in the detailed description and the drawings are to be taken merely as presently preferred examples and that the invention is limited only by the language of the claims.